CHAPTER 10

Volatile Use of Persistent Memory

Introduction

This chapter discusses how applications that require a large quantity of volatile memory can leverage high-capacity persistent memory as a complementary solution to dynamic random-access memory (DRAM).

Applications that work with large data sets, like in-memory databases, caching systems, and scientific simulations, are often limited by the amount of volatile memory capacity available in the system or the cost of the DRAM required to load a complete data set. Persistent memory provides a high capacity memory tier to solve these memory-hungry application problems.

In the memory-storage hierarchy (described in Chapter 1), data is stored in tiers with frequently accessed data placed in DRAM for low-latency access, and less frequently accessed data is placed in larger capacity, higher latency storage devices. Examples of such solutions include Redis on Flash (https://redislabs.com/redis-enterprise/technology/redis-on-flash/) and Extstore for Memcached (https://memcached.org/blog/extstore-cloud/).

For memory-hungy applications that do not require persistence, using the larger capacity persistent memory as volatile memory provides new opportunities and solutions.

Using persistent memory as a volatile memory solution is advantageous when an application:

- Has control over data placement between DRAM and other storage tiers within the system
- Does not need to persist data

S. Scargall, *Programming Persistent Memory*, https://doi.org/10.1007/978-1-4842-4932-1_10

- Can use the native latencies of persistent memory, which may be slower than DRAM but are faster than non-volatile memory express (NVMe) solid-state drives (SSDs).

Background

Applications manage different kinds of data structures such as user data, key-value stores, metadata, and working buffers. Architecting a solution that uses tiered memory and storage may enhance application performance, for example, placing objects that are accessed frequently and require low-latency access in DRAM while storing objects that require larger allocations that are not as latency-sensitive on persistent memory. Traditional storage devices are used to provide persistence.

Memory Allocation

As described in Chapters 1 through 3, persistent memory is exposed to the application using memory-mapped files on a persistent memory-aware file system that provides direct access to the application. Since `malloc()` and `free()` do not operate on different types of memory or memory-mapped files, an interface is needed that provides `malloc()` and `free()` semantics for multiple memory types. This interface is implemented as the memkind library (`http://memkind.github.io/memkind/`).

How it Works

The memkind library is a user-extensible heap manager built on top of `jemalloc`, which enables partitioning of the heap between multiple *kinds* of memory. Memkind was created to support different kinds of memory when high bandwidth memory (HBM) was introduced. A PMEM *kind* was introduced to support persistent memory.

Different "kinds" of memory are defined by the operating system memory policies that are applied to virtual address ranges. Memory characteristics supported by memkind without user extension include the control of non-uniform memory access (NUMA) and page sizes. Figure 10-1 shows an overview of libmemkind components and hardware support.

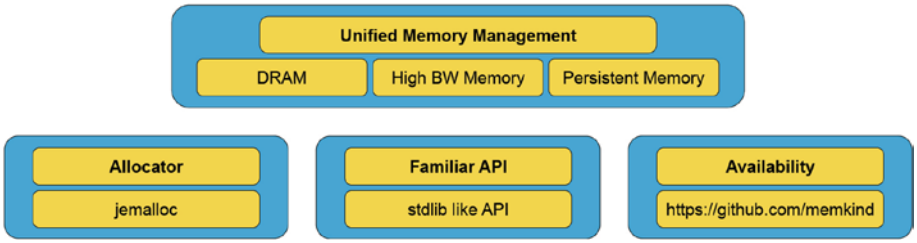

Figure 10-1. *An overview of the memkind components and hardware support*

The memkind library serves as a wrapper that redirects memory allocation requests from an application to an allocator that manages the heap. At the time of publication, only the `jemalloc` allocator is supported. Future versions may introduce and support multiple allocators. Memkind provides `jemalloc` with different kinds of memory: A *static kind* is created automatically, whereas a *dynamic kind* is created by an application using `memkind_create_kind()`.

Supported "Kinds" of Memory

The dynamic PMEM kind is best used with memory-addressable persistent storage through a DAX-enabled file system that supports load/store operations that are not paged via the system page cache. For the PMEM kind, the memkind library supports the traditional `malloc/free`-like interfaces on a memory-mapped file. When an application calls `memkind_create_kind()` with PMEM, a temporary file (`tmpfile(3)`) is created on a mounted DAX file system and is memory-mapped into the application's virtual address space. This temporary file is deleted automatically when the program terminates, giving the perception of volatility.

Figure 10-2 shows memory mappings from two memory sources: DRAM (`MEMKIND_DEFAULT`) and persistent memory (`PMEM_KIND`).

For allocations from DRAM, rather than using the common `malloc()`, the application can call `memkind_malloc()` with the *kind* argument set to `MEMKIND_DEFAULT`. `MEMKIND_DEFAULT` is a static kind that uses the operating system's default page size for allocations. Refer to the memkind documentation for large and huge page support.

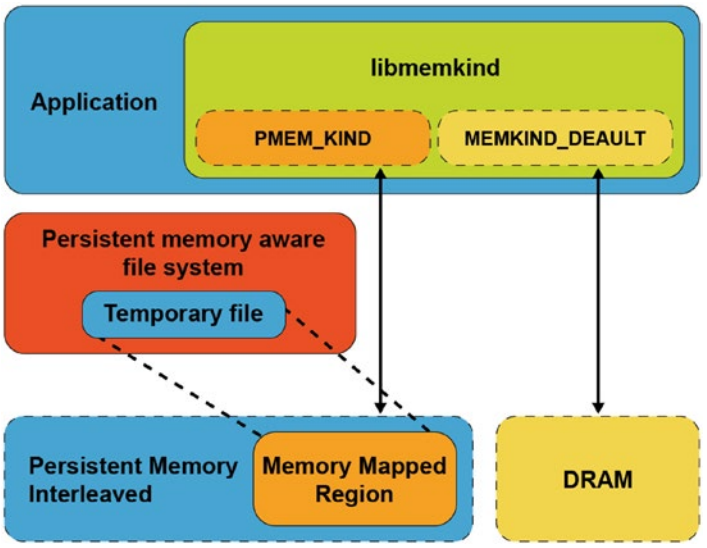

Figure 10-2. *An application using different "kinds" of memory*

When using `libmemkind` with DRAM and persistent memory, the key points to understand are:

- Two pools of memory are available to the application, one from DRAM and another from persistent memory.
- Both pools of memory can be accessed simultaneously by setting the kind type to `PMEM_KIND` to use persistent memory and `MEMKIND_DEFAULT` to use DRAM.
- `jemalloc` is the single memory allocator used to manage all kinds of memory.
- The memkind library is a wrapper around `jemalloc` that provides a unified API for allocations from different kinds of memory.
- `PMEM_KIND` memory allocations are provided by a temporary file (tmpfile(3)) created on a persistent memory-aware file system. The file is destroyed when the application exits. Allocations are not persistent.
- Using `libmemkind` for persistent memory requires simple modifications to the application.

The memkind API

The memkind API functions related to persistent memory programming are shown in Listing 10-1 and described in this section. The complete memkind API is available in the memkind man pages (http://memkind.github.io/memkind/man_pages/memkind.html).

Listing 10-1. Persistent memory-related memkind API functions

```
KIND CREATION MANAGEMENT:
int memkind_create_pmem(const char *dir, size_t max_size, memkind_t *kind);
int memkind_create_pmem_with_config(struct memkind_config *cfg, memkind_t
*kind);
memkind_t memkind_detect_kind(void *ptr);
int memkind_destroy_kind(memkind_t kind);

KIND HEAP MANAGEMENT:
void *memkind_malloc(memkind_t kind, size_t size);
void *memkind_calloc(memkind_t kind, size_t num, size_t size);
void *memkind_realloc(memkind_t kind, void *ptr, size_t size);
void memkind_free(memkind_t kind, void *ptr);
size_t memkind_malloc_usable_size(memkind_t kind, void *ptr);
memkind_t memkind_detect_kind(void *ptr);

KIND CONFIGURATION MANAGEMENT:
struct memkind_config *memkind_config_new();
void memkind_config_delete(struct memkind_config *cfg);
void memkind_config_set_path(struct memkind_config *cfg, const char
*pmem_dir);
void memkind_config_set_size(struct memkind_config *cfg, size_t pmem_size);
void memkind_config_set_memory_usage_policy(struct memkind_config *cfg,
memkind_mem_usage_policy policy);
```

Kind Management API

The memkind library supports a plug-in architecture to incorporate new memory kinds, which are referred to as dynamic kinds. The memkind library provides the API to create and manage the heap for the dynamic kinds.

Kind Creation

Use the `memkind_create_pmem()` function to create a PMEM *kind* of memory from a file-backed source. This file is created as a `tmpfile(3)` in a specified directory (`PMEM_DIR`) and is unlinked, so the file name is not listed under the directory. The temporary file is automatically removed when the program terminates.

Use `memkind_create_pmem()` to create a fixed or dynamic heap size depending on the application requirement. Additionally, configurations can be created and supplied rather than passing in configuration options to the `*_create_*` function.

Creating a Fixed-Size Heap

Applications that require a fixed amount of memory can specify a nonzero value for the `PMEM_MAX_SIZE` argument to `memkind_create_pmem()`, shown below. This defines the size of the memory pool to be created for the specified kind of memory. The value of `PMEM_MAX_SIZE` should be less than the available capacity of the file system specified in `PMEM_DIR` to avoid `ENOMEM` or `ENOSPC` errors. An internal data structure `struct memkind` is populated internally by the library and used by the memory management functions.

```
int memkind_create_pmem(PMEM_DIR, PMEM_MAX_SIZE, &pmem_kind)
```

The arguments to `memkind_create_pmem()` are

- `PMEM_DIR` is the directory where the temp file is created.
- `PMEM_MAX_SIZE` is the size, in bytes, of the memory region to be passed to `jemalloc`.
- `&pmem_kind` is the address of a memkind data structure.

If successful, `memkind_create_pmem()` returns zero. On failure, an error number is returned that `memkind_error_message()` can convert to an error message string. Listing 10-2 shows how a 32MiB PMEM kind is created on a `/daxfs` file system. Included in this listing is the definition of `memkind_fatal()` to print a memkind error message and exit. The rest of the examples in this chapter assume this routine is defined as shown below.

Listing 10-2. Creating a 32MiB PMEM kind

```
void memkind_fatal(int err)
{
    char error_message[MEMKIND_ERROR_MESSAGE_SIZE];
```

```
    memkind_error_message(err, error_message,
        MEMKIND_ERROR_MESSAGE_SIZE);
    fprintf(stderr, "%s\n", error_message);
    exit(1);
}

/* ... in main() ... */

#define PMEM_MAX_SIZE (1024 * 1024 * 32)

struct memkind *pmem_kind;
int err;

// Create PMEM memory pool with specific size
err = memkind_create_pmem("/daxfs",PMEM_MAX_SIZE, &pmem_kind);
if (err) {
    memkind_fatal(err);
}
```

You can also create a heap with a specific configuration using the function `memkind_create_pmem_with_config()`. This function uses a `memkind_config` structure with optional parameters such as size, file path, and memory usage policy. Listing 10-3 shows how to build a `test_cfg` using `memkind_config_new()`, then passing that configuration to `memkind_create_pmem_with_config()` to create a PMEM kind. We use the same `path` and `size` parameters from the Listing 10-2 example for comparison.

Listing 10-3. Creating PMEM kind with configuration

```
struct memkind_config *test_cfg = memkind_config_new();
memkind_config_set_path(test_cfg, "/daxfs");
memkind_config_set_size(test_cfg, 1024 * 1024 * 32);
memkind_config_set_memory_usage_policy(test_cfg, MEMKIND_MEM_USAGE_POLICY_
CONSERVATIVE);

// create a PMEM partition with specific configuration
err = memkind_create_pmem_with_config(test_cfg, &pmem_kind);
if (err) {
    memkind_fatal(err);
}
```

Creating a Variable Size Heap

When PMEM_MAX_SIZE is set to zero, as shown below, allocations are satisfied as long as the temporary file can grow. The maximum heap size growth is limited by the capacity of the file system mounted under the PMEM_DIR argument.

```
memkind_create_pmem(PMEM_DIR, 0, &pmem_kind)
```

The arguments to memkind_create_pmem() are:

- PMEM_DIR is the directory where the temp file is created.
- PMEM_MAX_SIZE is 0.
- &pmem_kind is the address of a memkind data structure.

If the PMEM kind is created successfully, memkind_create_pmem() returns zero. On failure, memkind_error_message() can be used to convert an error number returned by memkind_create_pmem() to an error message string, as shown in the memkind_fatal() routine in Listing 10-2.

Listing 10-4 shows how to create a PMEM kind with variable size.

Listing 10-4. Creating a PMEM kind with variable size

```
struct memkind *pmem_kind;
int err;
err = memkind_create_pmem("/daxfs",0,&pmem_kind);
if (err) {
    memkind_fatal(err);
}
```

Detecting the Memory Kind

Memkind supports both automatic detection of the kind as well as a function to detect the kind associated with a memory referenced by a pointer.

Automatic Kind Detection

Automatically detecting the kind of memory is supported to simplify code changes when using libmemkind. Thus, the memkind library will automatically retrieve the *kind* of memory pool the allocation was made from, so the heap management functions listed in Table 10-1 can be called without specifying the kind.

Table 10-1. *Automatic kind detection functions and their equivalent specified kind functions and operations*

Operation	Memkind API with Kind	Memkind API Using Automatic Detection
free	memkind_free(kind, ptr)	memkind_free(NULL, ptr)
realloc	memkind_realloc(kind, ptr, size)	memkind_realloc(NULL, ptr, size)
Get size of allocated memory	memkind_malloc_usable_size(kind, ptr)	memkind_malloc_usable_size(NULL, ptr)

The memkind library internally tracks the kind of a given object from the allocator metadata. However, to get this information, some of the operations may need to acquire a lock to prevent accesses from other threads, which may negatively affect the performance in a multithreaded environment.

Memory Kind Detection

Memkind also provides the `memkind_detect_kind()` function, shown below, to query and return the kind of memory referenced by the pointer passed into the function. If the input pointer argument is `NULL`, the function returns `NULL`. The input pointer argument passed into `memkind_detect_kind()` must have been returned by a previous call to `memkind_malloc()`, `memkind_calloc()`, `memkind_realloc()`, or `memkind_posix_memalign()`.

```
memkind_t memkind_detect_kind(void *ptr)
```

Similar to the automatic detection approach, this function has nontrivial performance overhead. Listing 10-5 shows how to detect the kind type.

Listing 10-5. pmem_detect_kind.c – how to automatically detect the 'kind' type

```
73  err = memkind_create_pmem(path, 0, &pmem_kind);
74  if (err) {
75      memkind_fatal(err);
76  }
77
```

```
78  /* do some allocations... */
79  buf0 = memkind_malloc(pmem_kind, 1000);
80  buf1 = memkind_malloc(MEMKIND_DEFAULT, 1000);
81
82  /* look up the kind of an allocation */
83  if (memkind_detect_kind(buf0) == MEMKIND_DEFAULT) {
84      printf("buf0 is DRAM\n");
85  } else {
86      printf("buf0 is pmem\n");
87  }
```

Destroying Kind Objects

Use the `memkind_destroy_kind()` function, shown below, to delete the kind object that was previously created using the `memkind_create_pmem()` or `memkind_create_pmem_with_config()` function.

```
int memkind_destroy_kind(memkind_t kind);
```

Using the same `pmem_detect_kind.c` code from Listing 10-5, Listing 10-6 shows how the kind is destroyed before the program exits.

Listing 10-6. Destroying a kind object

```
89      err = memkind_destroy_kind(pmem_kind);
90      if (err) {
91          memkind_fatal(err);
92      }
```

When the kind returned by `memkind_create_pmem()` or `memkind_create_pmem_with_config()` is successfully destroyed, all the allocated memory for the kind object is freed.

Heap Management API

The heap management functions described in this section have an interface modeled on the ISO C standard API, with an additional "kind" parameter to specify the memory type used for allocation.

Allocating Memory

The memkind library provides `memkind_malloc()`, `memkind_calloc()`, and `memkind_realloc()` functions for allocating memory, defined as follows:

```
void *memkind_malloc(memkind_t kind, size_t size);
void *memkind_calloc(memkind_t kind, size_t num, size_t size);
void *memkind_realloc(memkind_t kind, void *ptr, size_t size);
```

`memkind_malloc()` allocates size bytes of uninitialized memory of the specified kind. The allocated space is suitably aligned (after possible pointer coercion) for storage of any object type. If `size` is 0, then `memkind_malloc()` returns `NULL`.

`memkind_calloc()` allocates space for `num` objects, each is `size` bytes in length. The result is identical to calling `memkind_malloc()` with an argument of `num * size`. The exception is that the allocated memory is explicitly initialized to zero bytes. If `num` or `size` is 0, then `memkind_calloc()` returns `NULL`.

`memkind_realloc()` changes the size of the previously allocated memory referenced by `ptr` to size bytes of the specified kind. The contents of the memory remain unchanged, up to the lesser of the new and old sizes. If the new size is larger, the contents of the newly allocated portion of the memory are undefined. If successful, the memory referenced by `ptr` is freed, and a pointer to the newly allocated memory is returned.

The code example in Listing 10-7 shows how to allocate memory from DRAM and persistent memory (pmem_kind) using `memkind_malloc()`. Rather than using the common C library `malloc()` for DRAM and `memkind_malloc()` for persistent memory, we recommend using a single library to simplify the code.

Listing 10-7. An example of allocating memory from both DRAM and persistent memory

```
/*
 * Allocates 100 bytes using appropriate "kind"
 * of volatile memory
 */
```

```
// Create a PMEM memory pool with a specific size
  err = memkind_create_pmem(path, PMEM_MAX_SIZE, &pmem_kind);
  if (err) {
      memkind_fatal(err);
  }
  char *pstring = memkind_malloc(pmem_kind, 100);
  char *dstring = memkind_malloc(MEMKIND_DEFAULT, 100);
```

Freeing Allocated Memory

To avoid memory leaks, allocated memory can be freed using the `memkind_free()` function, defined as:

```
void memkind_free(memkind_t kind, void *ptr);
```

`memkind_free()` causes the allocated memory referenced by `ptr` to be made available for future allocations. This pointer must be returned by a previous call to `memkind_malloc()`, `memkind_calloc()`, `memkind_realloc()`, or `memkind_posix_memalign()`. Otherwise, if `memkind_free(kind, ptr)` was previously called, undefined behavior occurs. If `ptr` is `NULL`, no operation is performed. In cases where the `kind` is unknown in the context of the call to `memkind_free()`, `NULL` can be given as the kind specified to `memkind_free()`, but this will require an internal lookup for the correct kind. Always specify the correct kind because the lookup for kind could result in a serious performance penalty.

Listing 10-8 shows four examples of `memkind_free()` being used. The first two specify the kind, and the second two use `NULL` to detect the kind automatically.

Listing 10-8. Examples of memkind_free() usage

```
/* Free the memory by specifying the kind */
memkind_free(MEMKIND_DEFAULT, dstring);
memkind_free(PMEM_KIND, pstring);

/* Free the memory using automatic kind detection */
memkind_free(NULL, dstring);
memkind_free(NULL, pstring);
```

Kind Configuration Management

You can also create a heap with a specific configuration using the function `memkind_create_pmem_with_config()`. This function requires completing a `memkind_config` structure with optional parameters such as size, path to file, and memory usage policy.

Memory Usage Policy

In jemalloc, a runtime option called `dirty_decay_ms` determines how fast it returns unused memory back to the operating system. A shorter decay time purges unused memory pages faster, but the purging costs CPU cycles. Trade-offs between memory and CPU cycles needed for this operation should be carefully thought out before using this parameter.

The memkind library supports two policies related to this feature:

1. MEMKIND_MEM_USAGE_POLICY_DEFAULT
2. MEMKIND_MEM_USAGE_POLICY_CONSERVATIVE

The minimum and maximum values for `dirty_decay_ms` using the `MEMKIND_MEM_USAGE_POLICY_DEFAULT` are 0ms to 10,000ms for arenas assigned to a PMEM kind. Setting `MEMKIND_MEM_USAGE_POLICY_CONSERVATIVE` sets shorter decay times to purge unused memory faster, reducing memory usage. To define the memory usage policy, use `memkind_config_set_memory_usage_policy()`, shown below:

```
void memkind_config_set_memory_usage_policy (struct memkind_config *cfg,
memkind_mem_usage_policy policy );
```

- `MEMKIND_MEM_USAGE_POLICY_DEFAULT` is the default memory usage policy.
- `MEMKIND_MEM_USAGE_POLICY_CONSERVATIVE` allows changing the `dirty_decay_ms` parameter.

Listing 10-9 shows how to use `memkind_config_set_memory_usage_policy()` with a custom configuration.

Listing 10-9. An example of a custom configuration and memory policy use

```
73  struct memkind_config *test_cfg =
74      memkind_config_new();
75  if (test_cfg == NULL) {
76      fprintf(stderr,
77          "memkind_config_new: out of memory\n");
78      exit(1);
79  }
80
81  memkind_config_set_path(test_cfg, path);
82  memkind_config_set_size(test_cfg, PMEM_MAX_SIZE);
83  memkind_config_set_memory_usage_policy(test_cfg,
84      MEMKIND_MEM_USAGE_POLICY_CONSERVATIVE);
85
86  // Create PMEM partition with the configuration
87  err = memkind_create_pmem_with_config(test_cfg,
88      &pmem_kind);
89  if (err) {
90      memkind_fatal(err);
91  }
```

Additional memkind Code Examples

The memkind source tree contains many additional code examples, available on GitHub at https://github.com/memkind/memkind/tree/master/examples.

C++ Allocator for PMEM Kind

A new `pmem::allocator` class template is created to support allocations from persistent memory, which conforms to C++11 allocator requirements. It can be used with C++ compliant data structures from:

- Standard Template Library (STL)
- Intel® Threading Building Blocks (Intel® TBB) library

The pmem::allocator class template uses the memkind_create_pmem() function described previously. This allocator is stateful and has no default constructor.

pmem::allocator methods

```
pmem::allocator(const char *dir, size_t max_size);
pmem::allocator(const std::string& dir, size_t max_size) ;
template <typename U> pmem::allocator<T>::allocator(const
pmem::allocator<U>&);
template <typename U> pmem::allocator(allocator<U>&& other);
pmem::allocator<T>::~allocator();
T* pmem::allocator<T>::allocate(std::size_t n) const;
void pmem::allocator<T>::deallocate(T* p, std::size_t n) const ;
template <class U, class... Args> void pmem::allocator<T>::construct(U* p,
Args... args) const;
void pmem::allocator<T>::destroy(T* p) const;
```

For more information about the pmem::allocator class template, refer to the pmem allocator(3) man page.

Nested Containers

Multilevel containers such as a vector of lists, tuples, maps, strings, and so on pose challenges in handling the nested objects.

Imagine you need to create a vector of strings and store it in persistent memory. The challenges – and their solutions – for this task include:

1. Challenge: The std::string cannot be used for this purpose because it is an alias of the std::basic_string. The std::allocator requires a new alias that uses pmem:allocator.

   Solution: A new alias called pmem_string is defined as a typedef of std::basic_string when created with pmem::allocator.

2. Challenge: How to ensure that an outermost vector will properly construct nested `pmem_string` with a proper instance of `pmem::allocator`.

   Solution: From C++11 and later, the `std::scoped_allocator_adaptor` class template can be used with multilevel containers. The purpose of this adaptor is to correctly initialize stateful allocators in nested containers, such as when all levels of a nested container must be placed in the same memory segment.

C++ Examples

This section presents several full-code examples demonstrating the use of `libmemkind` using C and C++.

Using the pmem::allocator

As mentioned earlier, you can use `pmem::allocator` with any STL-like data structure. The code sample in Listing 10-10 includes a `pmem_allocator.h` header file to use `pmem::allocator`.

Listing 10-10. pmem_allocator.cpp: using pmem::allocator with std:vector

```
#include <pmem_allocator.h>
#include <vector>
#include <cassert>

int main(int argc, char *argv[]) {
    const size_t pmem_max_size = 64 * 1024 * 1024; //64 MB
    const std::string pmem_dir("/daxfs");

    // Create allocator object
    libmemkind::pmem::allocator<int>
        alc(pmem_dir, pmem_max_size);

```

```
49      // Create std::vector with our allocator.
50      std::vector<int,
51          libmemkind::pmem::allocator<int>> v(alc);
52
53      for (int i = 0; i < 100; ++i)
54          v.push_back(i);
55
56      for (int i = 0; i < 100; ++i)
57          assert(v[i] == i);
```

- Line 43: We define a persistent memory pool of 64MiB.
- Lines 46-47: We create an allocator object `alc` of type `pmem::allocator<int>`.
- Line 50: We create a vector object `v` of type `std::vector<int, pmem::allocator<int> >` and pass in the `alc` from line 47 object as an argument. The `pmem::allocator` is stateful and has no default constructor. This requires passing the allocator object to the vector constructor; otherwise, a compilation error occurs if the default constructor of `std::vector<int, pmem::allocator<int> >` is called because the vector constructor will try to call the default constructor of `pmem::allocator`, which does not exist yet.

Creating a Vector of Strings

Listing 10-11 shows how to create a vector of strings that resides in persistent memory. We define `pmem_string` as a `typedef` of `std::basic_string` with `pmem::allocator`. In this example, `std::scoped_allocator_adaptor` allows the vector to propagate the `pmem::allocator` instance to all `pmem_string` objects stored in the vector object.

Listing 10-11. vector_of_strings.cpp: creating a vector of strings

```
37  #include <pmem_allocator.h>
38  #include <vector>
39  #include <string>
40  #include <scoped_allocator>
41  #include <cassert>
```

```
42 #include <iostream>
43
44 typedef libmemkind::pmem::allocator<char> str_alloc_type;
45
46 typedef std::basic_string<char, std::char_traits<char>,
   str_alloc_type> pmem_string;
47
48 typedef libmemkind::pmem::allocator<pmem_string> vec_alloc_type;
49
50 typedef std::vector<pmem_string, std::scoped_allocator_adaptor
   <vec_alloc_type> > vector_type;
51
52 int main(int argc, char *argv[]) {
53     const size_t pmem_max_size = 64 * 1024 * 1024; //64 MB
54     const std::string pmem_dir("/daxfs");
55
56     // Create allocator object
57     vec_alloc_type alc(pmem_dir, pmem_max_size);
58     // Create std::vector with our allocator.
59     vector_type v(alc);
60
61     v.emplace_back("Foo");
62     v.emplace_back("Bar");
63
64     for (auto str : v) {
65             std::cout << str << std::endl;
66     }
```

- Line 46: We define `pmem_string` as a `typedef` of `std::basic_string`.
- Line 48: We define the `pmem::allocator` using the `pmem_string` type.
- Line 50: Using `std::scoped_allocator_adaptor` allows the `vector` to propagate the `pmem::allocator` instance to all `pmem_string` objects stored in the `vector` object.

Expanding Volatile Memory Using Persistent Memory

Persistent memory is treated by the kernel as a device. In a typical use-case, a persistent memory-aware file system is created and mounted with the *–o dax* option, and files are memory-mapped into the virtual address space of a process to give the application direct load/store access to persistent memory regions.

A new feature was added to the Linux kernel v5.1 such that persistent memory can be used more broadly as volatile memory. This is done by binding a persistent memory device to the kernel, and the kernel manages it as an extension to DRAM. Since persistent memory has different characteristics than DRAM, memory provided by this device is visible as a separate NUMA node on its corresponding socket.

To use the MEMKIND_DAX_KMEM kind, you need pmem to be available using *device DAX*, which exposes pmem as devices with names like /dev/dax*. If you have an existing dax device and want to migrate the device model type to use DEV_DAX_KMEM, use:

```
$ sudo daxctl migrate-device-model
```

To create a new dax device using all available capacity on the first available region (NUMA node), use:

```
$ sudo ndctl create-namespace --mode=devdax --map=mem
```

To create a new dax device specifying the region and capacity, use:

```
$ sudo ndctl create-namespace --mode=devdax --map=mem --region=region0
--size=32g
```

To display a list of namespaces, use:

```
$ ndctl list
```

If you have already created a namespace in another mode, such as the default fsdax, you can reconfigure the device using the following where `namespace0.0` is the existing namespace you want to reconfigure:

```
$ sudo ndctl create-namespace --mode=devdax --map=mem --force -e namespace0.0
```

For more details about creating new namespace read https://docs.pmem.io/ndctl-users-guide/managing-namespaces#creating-namespaces.

DAX devices must be converted to use the `system-ram` mode. Converting a dax device to a NUMA node suitable for use with system memory can be performed using following command:

```
$ sudo daxctl reconfigure-device dax2.0 --mode=system-ram
```

This will migrate the device from using the device_dax driver to the dax_pmem driver. The following shows an example output with dax1.0 configured as the default `devdax` type and `dax2.0` is `system-ram`:

```
$ daxctl list
    [
      {
        "chardev":"dax1.0",
        "size":263182090240,
        "target_node":3,
        "mode":"devdax"
      },
      {
        "chardev":"dax2.0",
        "size":263182090240,
        "target_node":4,
        "mode":"system-ram"
      }
    ]
```

You can now use numactl -H to show the hardware NUMA configuration. The following example output is collected from a 2-socket system and shows node 4 is a new system-ram backed NUMA node created from persistent memory:

```
$ numactl -H
    available: 3 nodes (0-1,4)
    node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
                 23 24 25 26 27 56 57 58 59 60 61 62 63 64 65 66 67 68 69
                 70 71 72 73 74 75 76 77 78 79 80 81 82 83
    node 0 size: 192112 MB
    node 0 free: 185575 MB
```

```
node 1 cpus: 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
             47 48 49 50 51 52 53 54 55 84 85 86 87 88 89 90 91 92 93
             94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109
             110 111
node 1 size: 193522 MB
node 1 free: 193107 MB
node 4 cpus:
node 4 size: 250880 MB
node 4 free: 250879 MB
node distances:
node   0   1   4
  0:  10  21  17
  1:  21  10  28
  4:  17  28  10
```

To online the NUMA node and have the Kernel manage the new memory, use:

```
$ sudo daxctl online-memory dax0.1
dax0.1: 5 sections already online
dax0.1: 0 new sections onlined
onlined memory for 1 device
```

At this point, the kernel will use the new capacity for normal operation. The new memory shows itself in tools such lsmem example shown below where we see an additional 10GiB of system-ram in the 0x0000003380000000-0x00000035ffffffff address range:

```
$ lsmem
RANGE                                  SIZE  STATE REMOVABLE   BLOCK
0x0000000000000000-0x000000007fffffff    2G online        no       0
0x0000000100000000-0x000000277fffffff  154G online       yes    2-78
0x0000002780000000-0x000000297fffffff    8G online        no   79-82
0x0000002980000000-0x0000002efffffffff  22G online       yes   83-93
0x0000002f00000000-0x0000002fffffffff    4G online        no   94-95
0x0000003380000000-0x00000035ffffffff   10G online       yes 103-107
0x000001aa80000000-0x000001d0ffffffff  154G online       yes 853-929
0x000001d100000000-0x000001d37fffffff   10G online        no 930-934
0x000001d380000000-0x000001d8ffffffff   22G online       yes 935-945
0x000001d900000000-0x000001d9ffffffff    4G online        no 946-947
```

```
Memory block size:         2G
Total online memory:     390G
Total offline memory:      0B
```

To programmatically allocate memory from a NUMA node created using persistent memory, a new static kind, called MEMKIND_DAX_KMEM, was added to `libmemkind` that uses the `system-ram` DAX device.

Using `MEMKIND_DAX_KMEM` as the first argument to memkind_malloc(), shown below, you can use persistent memory from separate NUMA nodes in a single application. The persistent memory is still physically connected to a CPU socket, so the application should take care to ensure CPU affinity for optimal performance.

```
memkind_malloc(MEMKIND_DAX_KMEM, size_t size)
```

Figure 10-3 shows an application that created two static kind objects: `MEMKIND_DEFAULT` and `MEMKIND_DAX_KMEM`.

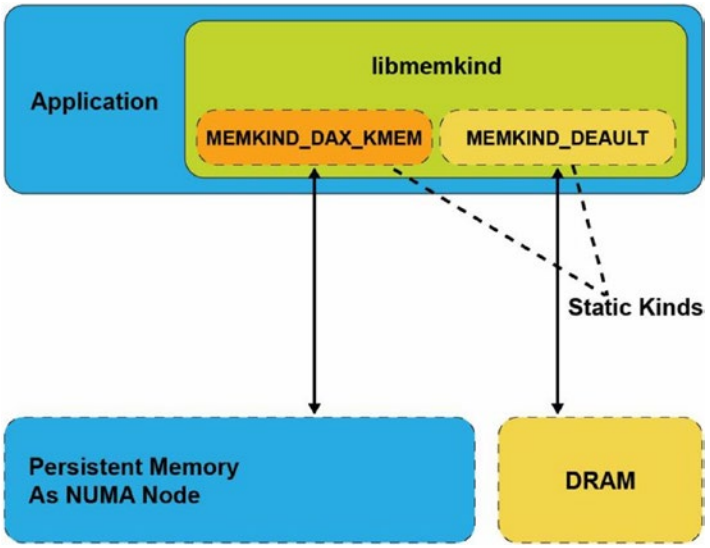

Figure 10-3. *An application that created two kind objects from different types of memory*

The difference between the PMEM_KIND described earlier and MEMKIND_DAX_KMEM is that the MEMKIND_DAX_KMEM is a static kind and uses mmap() with the MAP_PRIVATE flag, while the dynamic PMEM_KIND is created with memkind_create_pmem() and uses the MAP_SHARED flag when memory-mapping files on a DAX-enabled file system.

Child processes created using the `fork(2)` system call inherit the `MAP_PRIVATE` mappings from the parent process. When memory pages are modified by the parent process, a copy-on-write mechanism is triggered by the kernel to create an unmodified copy for the child process. These pages are allocated on the same NUMA node as the original page.

libvmemcache: An Efficient Volatile Key-Value Cache for Large-Capacity Persistent Memory

Some existing in-memory databases (IMDB) rely on manual dynamic memory allocations (`malloc, jemalloc, tcmalloc`), which can exhibit external and internal memory fragmentation when run for a long period of time, leaving large amounts of memory un-allocatable. Internal and external fragmentation is briefly explained as follows:

- *Internal fragmentation* occurs when more memory is allocated than is required, and the unused memory is contained within the allocated region. For example, if the requested allocation size is 200 bytes, a chunk of 256 bytes is allocated.
- *External fragmentation* occurs when variable memory sizes are allocated dynamically, resulting in a failure to allocate a contiguous chunk of memory, although the requested chunk of memory remains available in the system. This problem is more pronounced when large capacities of persistent memory are being used as volatile memory. Applications with substantially long runtimes need to solve this problem, especially if the allocated sizes have considerable variation. Applications and runtime environments handle this problem in different ways, for example:
    - Java and .NET use compacting garbage collection
    - Redis and Apache Ignite* use defragmentation algorithms
    - Memcached uses a slab allocator

Each of the above allocator mechanisms has pros and cons. Garbage collection and defragmentation algorithms require processing to occur on the heap to free unused allocations or move data to create contiguous space. Slab allocators usually define a fixed set of different sized buckets at initialization without knowing how many of each bucket

the application will need. If the slab allocator depletes a certain bucket size, it allocates from larger sized buckets, which reduces the amount of free space. These mechanisms can potentially block the application's processing and reduce its performance.

libvmemcache Overview

`libvmemcache` is an embeddable and lightweight in-memory caching solution with a key-value store at its core. It is designed to take full advantage of large-capacity memory, such as persistent memory, efficiently using memory mapping in a scalable way. It is optimized for use with memory-addressable persistent storage through a DAX-enabled file system that supports load/store operations. `libvmemcache` has these unique characteristics:

- The extent-based memory allocator sidesteps the fragmentation problem that affects most in-memory databases, and it allows the cache to achieve very high space utilization for most workloads.
- Buffered LRU (least recently used) combines a traditional LRU doubly linked list with a non-blocking ring buffer to deliver high scalability on modern multicore CPUs.
- A unique indexing `critnib` data structure delivers high performance and is very space efficient.

The cache for `libvmemcache` is tuned to work optimally with relatively large value sizes. While the smallest possible size is 256 bytes, `libvmemcache` performs best if the expected value sizes are above 1 kilobyte.

`libvmemcache` has more control over the allocation because it implements a custom memory-allocation scheme using an extents-based approach (like that of file system extents). `libvmemcache` can, therefore, concatenate and achieve substantial space efficiency. Additionally, because it is a cache, it can evict data to allocate new entries in a worst-case scenario. `libvmemcache` will *always* allocate exactly as much memory as it freed, minus metadata overhead. This is not true for caches based on common memory allocators such as memkind. `libvmemcache` is designed to work with terabyte-sized in-memory workloads, with very high space utilization.

`libvmemcache` works by automatically creating a temporary file on a DAX-enabled file system and memory-mapping it into the application's virtual address space. The temporary file is deleted when the program terminates and gives the perception of volatility. Figure 10-4 shows the application using traditional `malloc()` to allocate memory from DRAM and using `libvmemcache` to memory map a temporary file residing on a DAX-enabled file system from persistent memory.

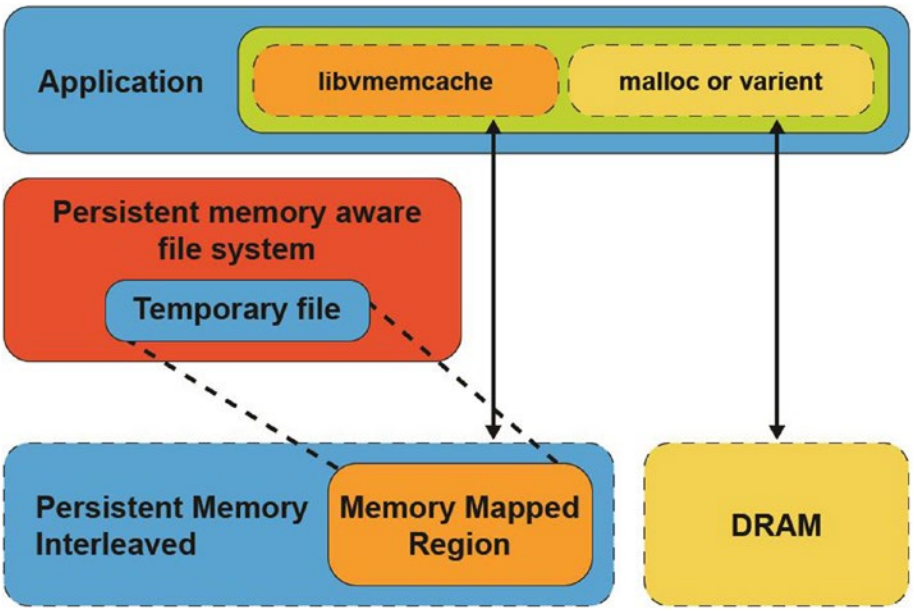

Figure 10-4. *An application using libvmemcache memory-maps a temporary file from a DAX-enabled file system*

Although libmemkind supports different kinds of memory and memory consumption policies, the underlying allocator is `jemalloc`, which uses dynamic memory allocation. Table 10-2 compares the implementation details of libvmemcache and libmemkind.

Table 10-2. *Design aspects of libmemkind and libvmemcache*

	libmemkind (PMEM)	libvmemcache
Allocation Scheme	Dynamic allocator	Extent based (not restricted to sector, page, etc.)
Purpose	General purpose	Lightweight in-memory cache
Fragmentation	Apps with random size allocations/ deallocations that run for a longer period	Minimized

libvmemcache Design

libvmemcache has two main design aspects:

1. Allocator design to improve/resolve fragmentation issues
2. A scalable and efficient LRU policy

Extent-Based Allocator

`libvmemcache` can solve fragmentation issues when working with terabyte-sized in-memory workloads and provide high space utilization. Figure 10-5 shows a workload example that creates many small objects, and over time, the allocator stops due to fragmentation.

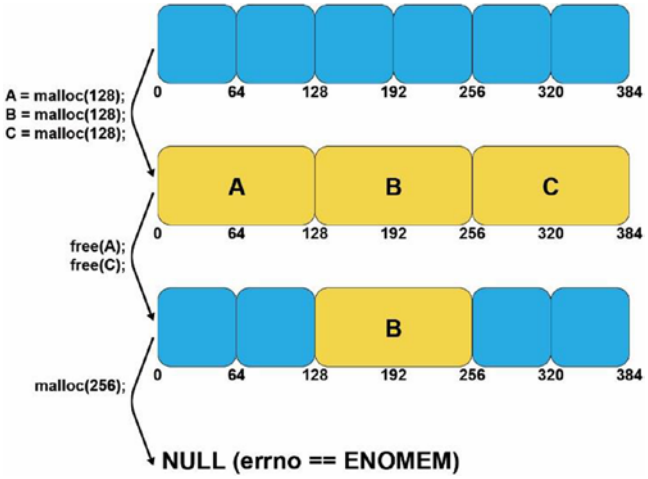

Figure 10-5. *An example of a workload that creates many small objects, and the allocator stops due to fragmentation*

`libvmemcache` uses an extent-based allocator, where an extent is a contiguous set of blocks allocated for storing the data in a database. Extents are typically used with large blocks supported by file systems (sectors, pages, etc.), but such restrictions do not apply when working with persistent memory that supports smaller block sizes (cache line). Figure 10-6 shows that if a single contiguous free block is not available to allocate an object, multiple, noncontiguous blocks are used to satisfy the allocation request. The noncontiguous allocations appear as a single allocation to the application.

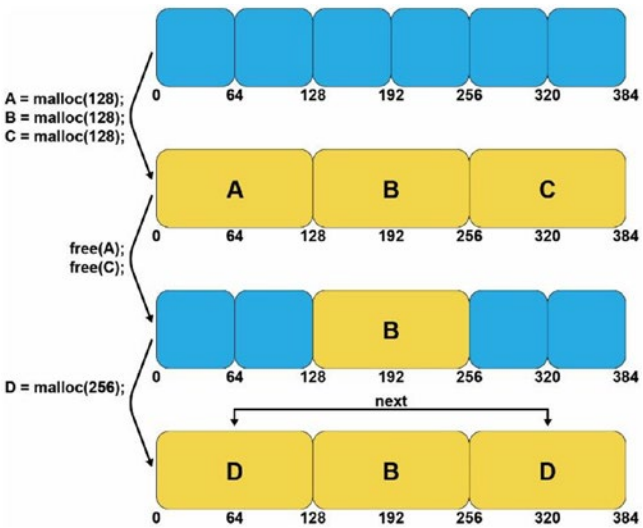

Figure 10-6. *Using noncontiguous free blocks to fulfill a larger allocation request*

Scalable Replacement Policy

An LRU cache is traditionally implemented as a doubly linked list. When an item is retrieved from this list, it gets moved from the middle to the front of the list, so it is not evicted. In a multithreaded environment, multiple threads may contend with the front element, all trying to move elements being retrieved to the front. Therefore, the front element is always locked (along with other locks) before moving the element being retrieved, which results in lock contention. This method is not scalable and is inefficient.

A buffer-based LRU policy creates a scalable and efficient replacement policy. A non-blocking ring buffer is placed in front of the LRU linked list to track the elements being retrieved. When an element is retrieved, it is added to this buffer, and only when the buffer is full (or the element is being evicted), the linked list is locked, and the elements in that buffer are processed and moved to the front of the list. This method preserves the LRU policy and provides a scalable LRU mechanism with minimal performance impact. Figure 10-7 shows a ring buffer-based design for the LRU algorithm.

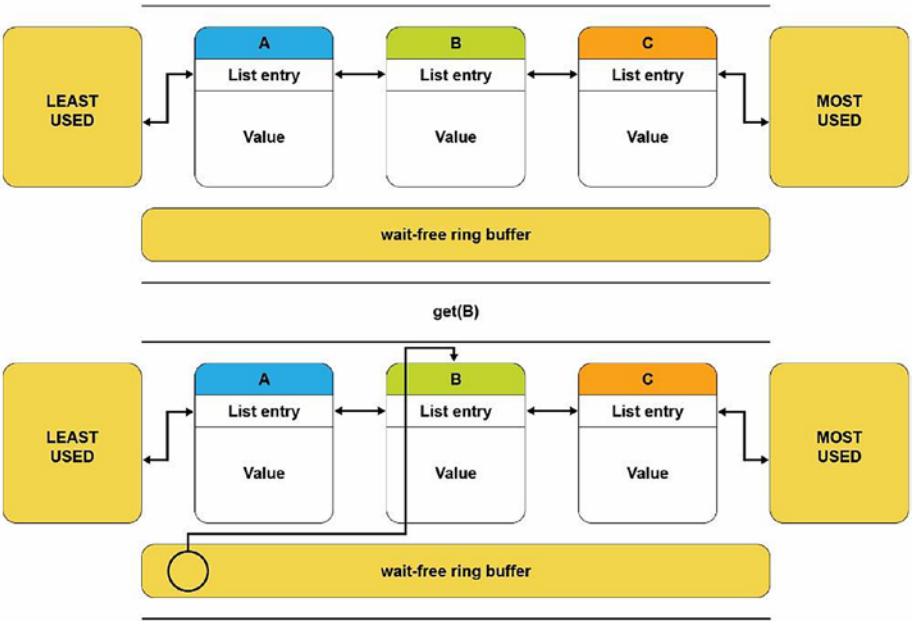

Figure 10-7. *A ring buffer-based LRU design*

Using libvmemcache

Table 10-3 lists the basic functions that libvmemcache provides. For a complete list, see the `libvmemcache` man pages (`https://pmem.io/vmemcache/manpages/master/vmemcache.3.html`).

Table 10-3. *The libvmemcache functions*

Function Name	Description
vmemcache_new	Creates an empty unconfigured vmemcache instance with default values: Eviction_policy=VMEMCACHE_REPLACEMENT_LRU Extent_size = VMEMCAHE_MIN_EXTENT VMEMCACHE_MIN_POOL
vmemcache_add	Associates the cache with a path.
vmemcache_set_size	Sets the size of the cache.
vmemcache_set_extent_size	Sets the block size of the cache (256 bytes minimum).
vmemcache_set_eviction_policy	Sets the eviction policy: 1. VMEMCACHE_REPLACEMENT_NONE 2. VMEMCACHE_REPLACEMENT_LRU
vmemcache_add	Associates the cache with a given path on a DAX-enabled file system or non-DAX-enabled file system.
vmemcache_delete	Frees any structures associated with the cache.
vmemcache_get	Searches for an entry with the given key, and if found, the entry's value is copied to `vbuf`.
vmemcache_put	Inserts the given `key-value pair` into the cache.
vmemcache_evict	Removes the given key from the cache.
vmemcache_callback_on_evict	Called when an entry is being removed from the cache.
vmemcache_callback_on_miss	Called when a get query fails to provide an opportunity to insert the missing key.

To illustrate how libvmemcache is used, Listing 10-12 shows how to create an instance of vmemcache using default values. This example uses a temporary file on a DAX-enabled file system and shows how a callback is registered after a cache miss for a key "meow."

Listing 10-12. vmemcache.c: An example program using libvmemcache

```
#include <libvmemcache.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STR_AND_LEN(x) (x), strlen(x)

VMEMcache *cache;

void on_miss(VMEMcache *cache, const void *key,
    size_t key_size, void *arg)
{
    vmemcache_put(cache, STR_AND_LEN("meow"),
         STR_AND_LEN("Cthulhu fthagn"));
}

void get(const char *key)
{
    char buf[128];
    ssize_t len = vmemcache_get(cache,
    STR_AND_LEN(key), buf, sizeof(buf), 0, NULL);
    if (len >= 0)
        printf("%.*s\n", (int)len, buf);
    else
        printf("(key not found: %s)\n", key);
}

int main()
{
```

```
66      cache = vmemcache_new();
67      if (vmemcache_add(cache, "/daxfs")) {
68          fprintf(stderr, "error: vmemcache_add: %s\n",
69                  vmemcache_errormsg());
70              exit(1);
71      }
72
73      // Query a non-existent key
74      get("meow");
75
76      // Insert then query
77      vmemcache_put(cache, STR_AND_LEN("bark"),
78          STR_AND_LEN("Lorem ipsum"));
79      get("bark");
80
81      // Install an on-miss handler
82      vmemcache_callback_on_miss(cache, on_miss, 0);
83      get("meow");
84
85      vmemcache_delete(cache);
```

- Line 66: Creates a new instance of `vmemcache` with default values for `eviction_policy and extent_size`.
- Line 67: Calls the `vmemcache_add()` function to associate cache with a given path.
- Line 74: Calls the `get()` function to query on an existing key. This function calls the `vmemcache_get()` function with error checking for success/failure of the function.
- Line 77: Calls `vmemcache_put()` to insert a new key.
- Line 82: Adds an on-miss callback handler to insert the key “meow” into the cache.
- Line 83: Retrieves the key “meow” using the `get()` function.
- Line 85: Deletes the `vmemcache` instance.

Summary

This chapter showed how persistent memory's large capacity can be used to hold volatile application data. Applications can choose to allocate and access data from DRAM or persistent memory or both.

memkind is a very flexible and easy-to-use library with semantics that are similar to the `libc malloc/free` APIs that developers frequently use.

`libvmemcache` is an embeddable and lightweight in-memory caching solution that allows applications to efficiently use persistent memory's large capacity in a scalable way. `libvmemcache` is an open source project available on GitHub at https://github.com/pmem/vmemcache.